19 | 生成订单信息之一：应用JDBC池优化和内存溢出分析

2021-05-03 高楼

高楼的性能工程实战课 进入课程 >

讲述：高楼
时长 17:10 大小 15.74M

你好，我是高楼。

在这节课中，我们来看一下生成订单接口的基准场景是什么结果。

你将看到一些重复的问题，比如 SQL 的问题定位，虽然具体的问题不同，但我们的分析逻辑没有区别，我会简单带过。同时，你也会看到一些新的问题，比如 JDBC 池增加之后，由于数据量过大导致 JVM 内存被消耗光；批量业务和实时业务共存导致的锁问题等。这节课，我们重点来看看这样的问题如何进一步优化。

话不多说，开整！

场景运行数据

对于生成订单接口，我们第一次试执行性能场景的结果如下：

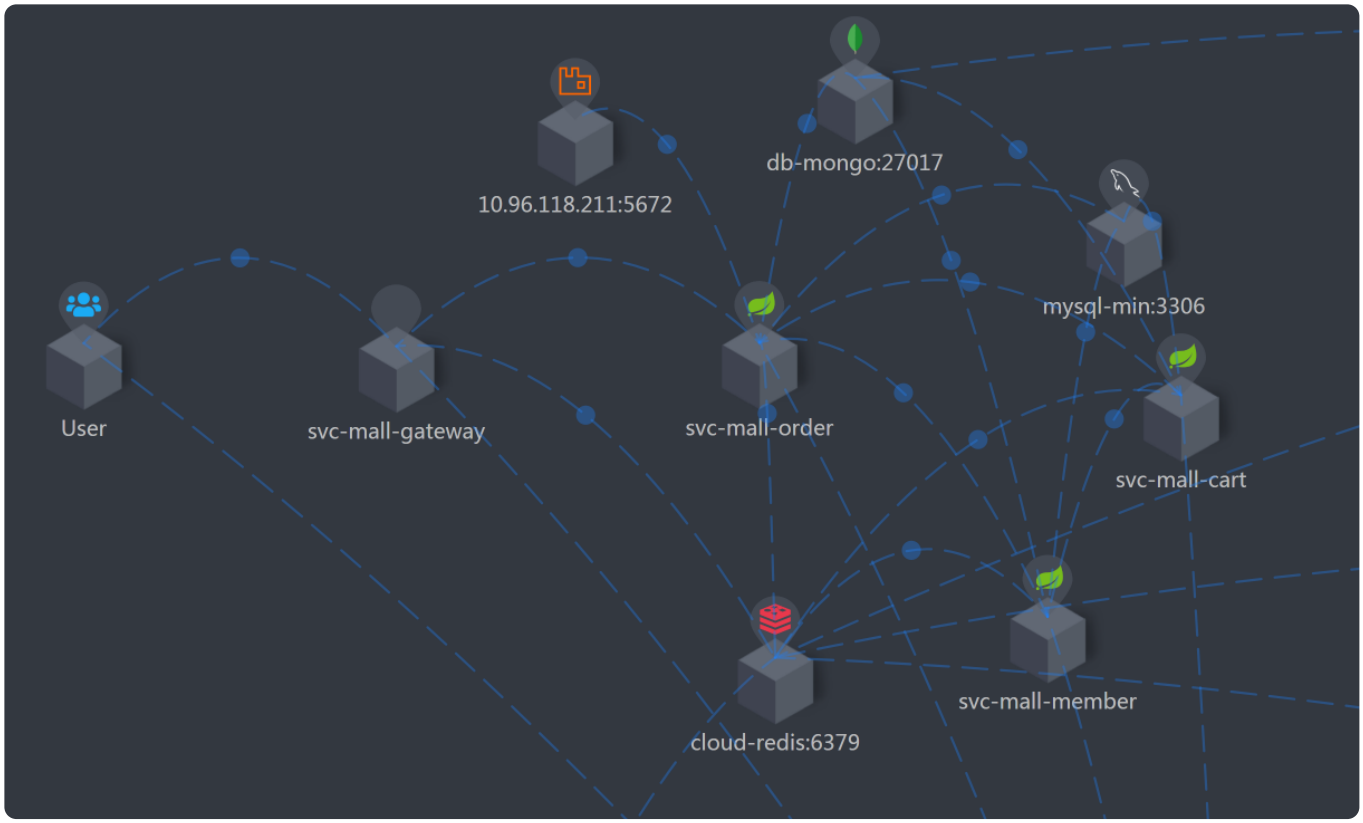
从场景执行的结果来看。40 个压力线程只跑出来 50 多的 TPS，响应时间也蹭蹭蹭地跑了近 700ms。这显然是一个非常慢的接口了。

从这样的接口来看，我们选择这个项目算是选择对了，因为到处都是性能问题。

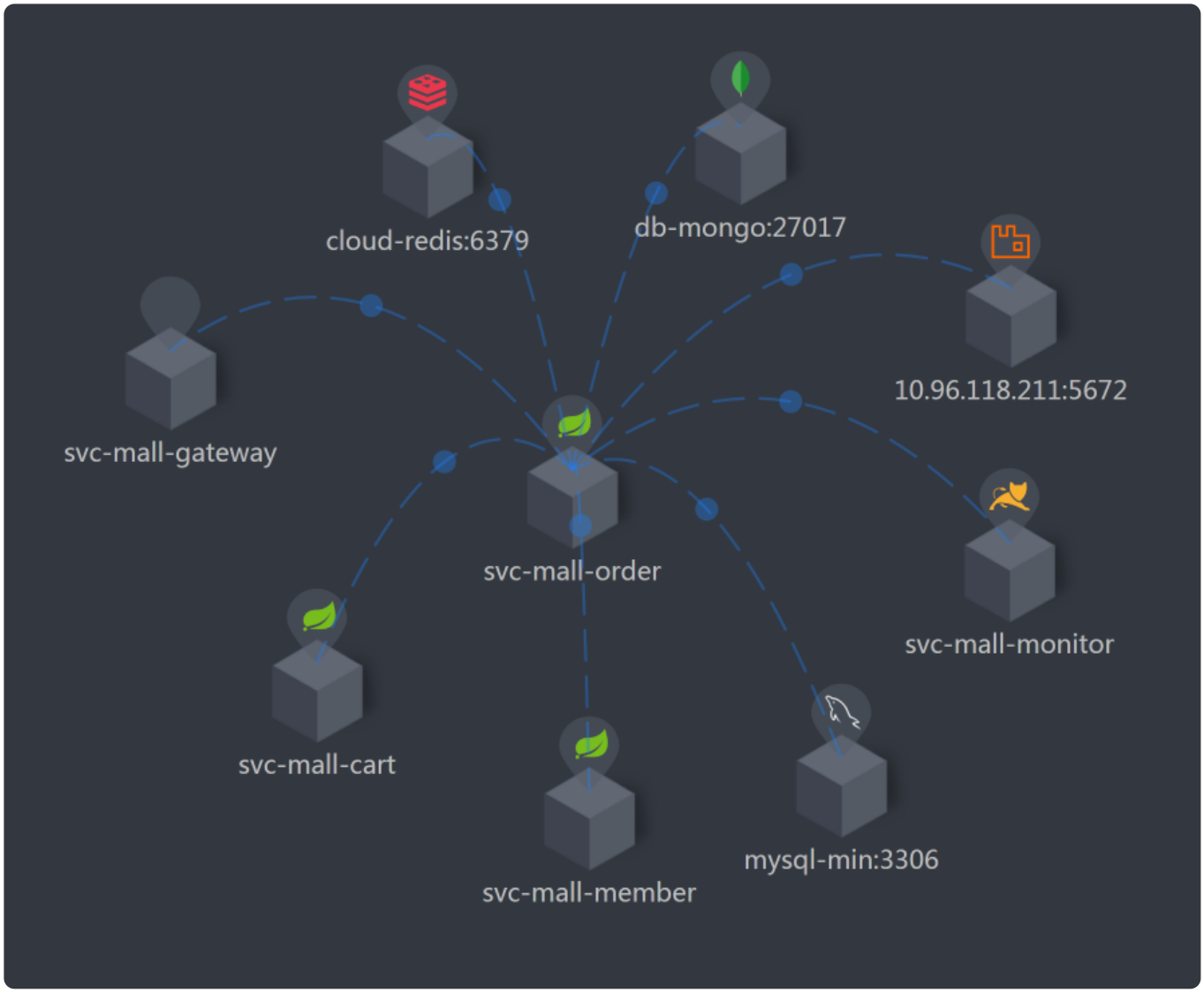
下面我们就来一步步分析一下。

架构图

前面我们做过多次描述，画架构图是为了知道分析的路径。所以按照惯例，我们仍然把架构图列在这里。



由于这个接口比较复杂，架构图看起来有点乱，我又整了一个简化版本：



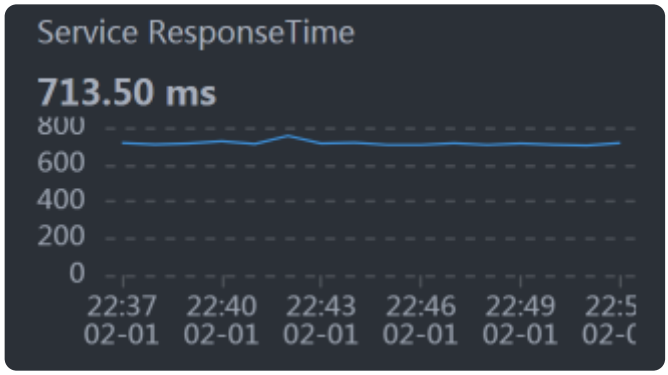
Order 服务是这个接口的核心，因此，你可以看到我把 Order 相关的服务都筛选了出来，这样我们就能很清楚地知道它连接了哪些东西。

下面我们来拆分响应时间。

拆分响应时间

因为在场景运行的时候，我们看到响应时间比较长，所以我们用 APM 工具来拆分一下：

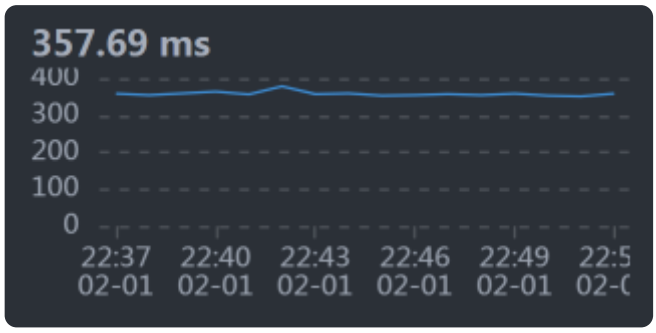
Gateway ：



从上图我们就可以看到 Gateway 上的时间在 700ms 左右，这与前面的场景数据是可以对上的。

我说明一下，这张小图的采样间隔是分钟，因此，你可能会发现这个曲线和压力工具给出的 TPS 曲线，在一些细节上对应不起来。不过这没关系，我们更应该关注整体的趋势。

Order：



我们前面提到，Order 是生产订单信息这个接口的重点，并且它的业务逻辑也非常复杂，因此，我们要多关注这个服务。

从数据上来看，Order 的是时间消耗在 350 毫秒左右，占到整个响应时间的一半。这是我们着重要分析的，而 Gateway 的转发能力也是要考虑的问题点，只是 Gateway 上没有逻辑，只做转发，如果是因为数据量大而导致的 Gateway 转发慢，那我们解决了 Order 的问题之后，Gateway 的问题也就会被解决。所以，我们先分析 Order 的问题。

所以，我们现在就来分析一下。

第一阶段

全局监控分析

我们先看全局监控：

服务器资源总览表【JOB：node-exporter，主机总数：12】

IP（链接到明细）	主机名	运行时间	内存	CPU核	5m负载	CPU使用率	内存使用率	分区使用率*	磁盘读取*	磁盘写入*	连接数	TCP_tw	下载带宽*	上传带宽*
k8s-master-1	k8s-master-1	3.73 week	7.64 GiB	4	0.32	14.65%	23.05%	29.95%	0 B/s	92.52 kB/s	282	43	566.16 kbps	395.94 kbps
k8s-master-2	k8s-master-2	6.31 week	7.64 GiB	4	1.15	20.21%	25.50%	29.95%	0 B/s	93.33 kB/s	318	33	892.27 kbps	892.27 kbps
k8s-master-3	k8s-master-3	6.31 week	7.64 GiB	4	0.49	14.24%	24.26%	29.95%	0 B/s	92.43 kB/s	294	105	265.64 kbps	538.47 kbps
k8s-worker-1	k8s-worker-1	5.07 day	15.51 GiB	6	0.77	14.53%	42.55%	64.72%	0 B/s	2.00 MB/s	28	72	50.86 Mbps	51.03 Mbps
k8s-worker-2	k8s-worker-2	1.69 hour	15.51 GiB	6	0.14	4.00%	11.19%	29.95%	4.60 kB/s	1.02 kB/s	27	78	33.59 kbps	28.08 kbps
k8s-worker-3	k8s-worker-3	2.78 week	15.51 GiB	6	6.03	55.46%	62.74%	50.38%	23.02 kB/s	53.17 kB/s	30	290	47.78 Mbps	47.66 Mbps
k8s-worker-4	k8s-worker-4	5.22 week	15.51 GiB	6	6.41	52.68%	41.52%	46.29%	23.62 kB/s	49.07 kB/s	55	8 K	2.91 Mbps	4.08 Mbps
k8s-worker-5	k8s-worker-5	5.22 week	15.51 GiB	6	0.77	14.89%	27.47%	36.48%	0 B/s	2.46 kB/s	28	118	1.55 Mbps	1.89 Mbps
k8s-worker-6	k8s-worker-6	5.22 week	15.51 GiB	6	0.88	13.66%	21.44%	39.35%	0 B/s	3.71 kB/s	30	111	851.03 kbps	823.51 kbps
k8s-worker-7	k8s-worker-7	4.92 week	15.51 GiB	4	0.84	18.79%	30.47%	56.13%	9.21 kB/s	27.69 kB/s	29	272	4.79 Mbps	3.88 Mbps
k8s-worker-8	k8s-worker-8	3.74 week	15.51 GiB	4	3.18	32.57%	61.64%	58.95%	0 B/s	73.61 kB/s	28	397	8.59 Mbps	5.43 Mbps
k8s-worker-9	k8s-worker-9	4.92 week	15.51 GiB	6	0.18	5.52%	26.77%	31.97%	18.41 kB/s	6.98 kB/s	28	79	1.23 Mbps	2.30 Mbps

一眼扫过去，啥也没有。既没有明显的 CPU 资源消耗，也没有明显的网络资源、IO 资源瓶颈。

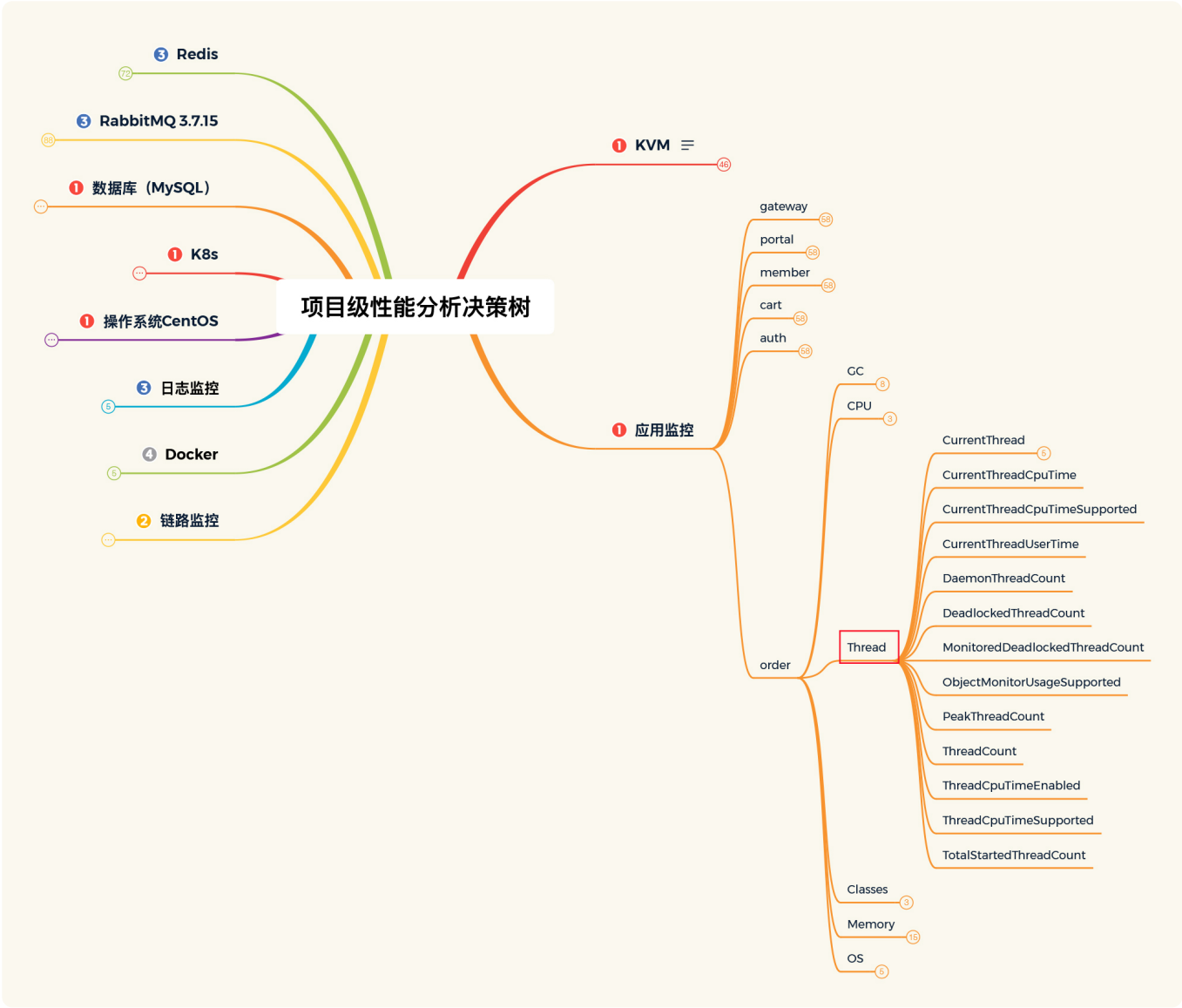
遇到这种情况，我们一定要留意整个链路上有限制的点。什么是有限制的点？比如说，各种池（连接池、等）、栈中的锁、数据库连接、还有数据库的锁之类。其实，总结下来就是一个关键词：**阻塞。**

我们只要分析出阻塞的点，就能把链路扩宽，进而把资源都用起来。

当然，也有可能在你分析了一圈之后，发现没有任何有阻塞的点，可是资源就是用不上去。这种情况只有一种可能，那就是你分析得还不够细致。因为可能存在阻塞的地方实在太多了，我们只能一步步拆解。

定向监控分析

正所谓“心中常备决策树，让你分析不迷路”。到了定向监控分析这里，按照第 4 讲中强调的性能分析决策树，我们先来分析 Order 服务：



在我分析 Order 的线程栈信息时，发现在 Order 的栈中，有大量这样的内容：

复制代码

```
"http-nio-8086-exec-421" Id=138560 WAITING on java.util.concurrent.locks.Abstr
    at sun.misc.Unsafe.park(Native Method)
    -  waiting on java.util.concurrent.locks.AbstractQueuedSynchronizer$Condit
    at java.util.concurrent.locks.LockSupport.park(LockSupport.java:175)
    at java.util.concurrent.locks.AbstractQueuedSynchronizer$ConditionObject.a
    at com.alibaba.druid.pool.DruidDataSource.takeLast(DruidDataSource.java:18
    at com.alibaba.druid.pool.DruidDataSource.getConnectionInternal(DruidDataS
    at com.alibaba.druid.pool.DruidDataSource.getConnectionDirect(DruidDataSou
    at com.alibaba.druid.filter.FilterChainImpl.dataSource_connect(FilterChain
    at com.alibaba.druid.filter.stat.StatFilter.dataSource_getConnection(StatF
    at com.alibaba.druid.filter.FilterChainImpl.dataSource_connect(FilterChain
    at com.alibaba.druid.pool.DruidDataSource.getConnection(DruidDataSource.ja
    at com.alibaba.druid.pool.DruidDataSource.getConnection(DruidDataSource.ja
    at com.alibaba.druid.pool.DruidDataSource.getConnection(DruidDataSource.ja
    ..........................
    at com.dunshan.mall.order.service.impl.PortalOrderServiceImpl$$EnhancerByS
    at com.dunshan.mall.order.controller.PortalOrderController.generateOrder$o
```

```
18      at com.dunshan.mall.order.controller.PortalOrderController.generateOrder$o
19      at com.dunshan.mall.order.controller.PortalOrderController$auxiliary$MTWkG
20      ..........................
21      at com.dunshan.mall.order.controller.PortalOrderController.generateOrder(P
22
```

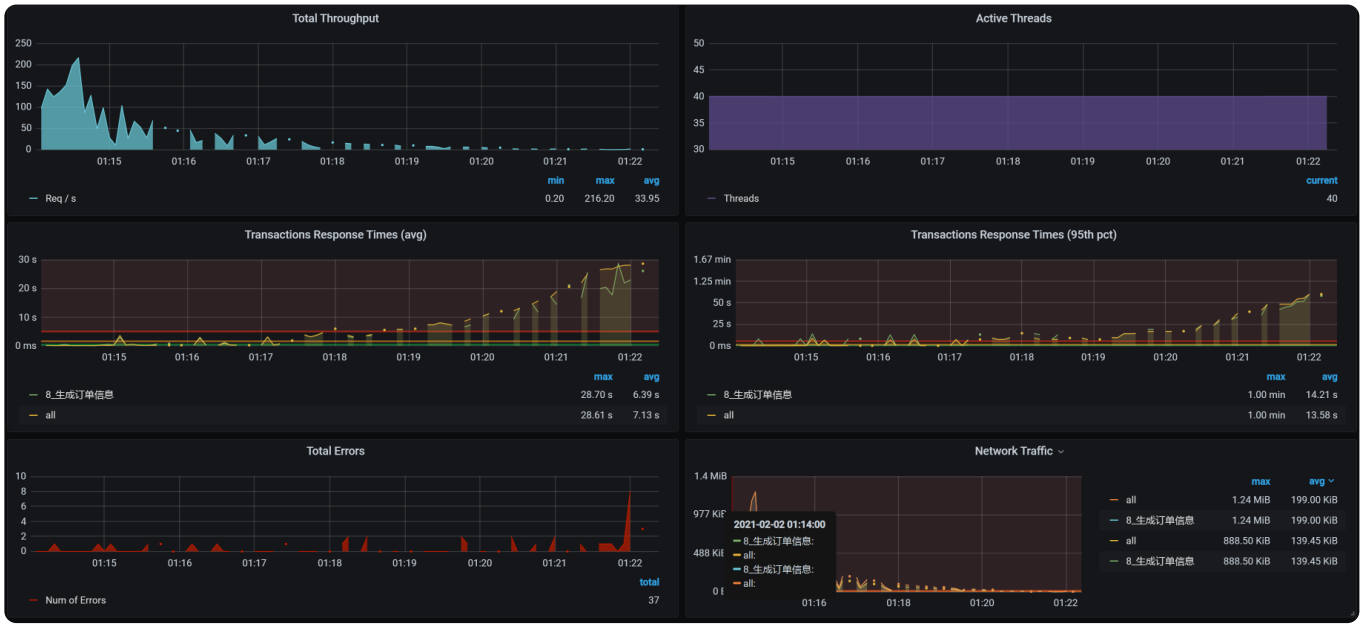
你看，栈信息中有很多 getConnection，这明显是 Order 服务在等数据库连接池。所以，我们要做的就是把 JDBC 池加大：

复制代码

```
1  原配置：
2         initial-size: 5 #连接池初始化大小
3         min-idle: 10 #最小空闲连接数
4         max-active: 20 #最大连接数
5
6
7  修改为：
8         initial-size: 20 #连接池初始化大小
9         min-idle: 10 #最小空闲连接数
10        max-active: 40 #最大连接数
```

你可以看到，我在这里并没有把 JDBC 池一次性修改得太大，主要是因为我不想为了维护连接池而产生过多的 CPU 消耗。我也建议你在增加资源池的时候，先一点点增加，看看有没有效果，等有了效果后再接着增加。

修改 JDBC 池后，我们再来看一下压力场景的执行数据：



从数据上看，TPS 有上升的趋势，并且一度达到了 150 以上。可是紧接着，TPS 就掉下来了，这个时候的响应时间倒是没有明显增加。而且你看，TPS 不仅掉下来了，而且还断断续续的，极为不稳定。

此外，我们还可以发现，在后续的压力中不仅有错误信息产生，响应时间也在上升。与此同时，我查看了全局监控的资源，并没有发现太大的资源消耗。既然有错误产生，没二话，我们只能整它！

第二阶段

全局监控分析

因为我们在前面修改了 Order 的 JDBC 池，所以在出现新的问题之后，我们先来看一下 Order 服务的健康状态。在查看 Order 服务的 top 时，看到如下信息：

复制代码

```
top - 01:28:17 up 19 days, 11:54,  3 users,  load average: 1.14, 1.73, 2.27
Tasks: 316 total,   1 running, 315 sleeping,   0 stopped,   0 zombie
%Cpu0  :100.0 us,  0.0 sy,  0.0 ni,  0.0 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu1  :  3.0 us,  2.7 sy,  0.0 ni, 93.6 id,  0.0 wa,  0.0 hi,  0.3 si,  0.3 s
%Cpu2  :  3.4 us,  3.4 sy,  0.0 ni, 93.3 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu3  :  3.7 us,  2.8 sy,  0.0 ni, 93.5 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
%Cpu4  :  3.6 us,  2.1 sy,  0.0 ni, 93.6 id,  0.0 wa,  0.0 hi,  0.3 si,  0.3 s
%Cpu5  :  2.8 us,  1.8 sy,  0.0 ni, 95.4 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0 s
KiB Mem : 16265992 total,  2229060 free,  9794944 used,  4241988 buff/cache
KiB Swap:        0 total,        0 free,        0 used.  6052732 avail Mem


  PID USER      PR  NI    VIRT    RES    SHR S  %CPU %MEM     TIME+ COMMAND
29349 root      20   0 8836040   4.3g  16828 S  99.7 27.4  20:51.90 java
 1089 root      20   0 2574864  98144  23788 S   6.6  0.6   2066:38 kubelet
```

悲催的数据还是来了，你看，有一个 us cpu 达到了 100%！这是啥情况?

我通过 top -Hp 和 jstack -l 1 两个命令查看进程后发现，原来是 VM Thread 线程占用了 CPU，这个线程是做垃圾回收（GC）的。 既然如此，那我们就来看一下内存的回收状态，查看 jstat 如下：

复制代码

```
[root@svc-mall-order-7fbdd7b85f-ks828 /]# jstat -gcutil 1 1s
```

```
  S0     S1     E      O      M      CCS    YGC    YGCT    FGC    FGCT     GCT
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     93  652.664  681.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     93  652.664  681.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     93  652.664  681.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     93  652.664  681.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     93  652.664  681.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     93  652.664  681.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     93  652.664  681.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     94  659.863  688.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     94  659.863  688.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     94  659.863  688.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     94  659.863  688.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     94  659.863  688.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     94  659.863  688.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     94  659.863  688.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     95  667.472  696.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     95  667.472  696.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     95  667.472  696.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     95  667.472  696.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     95  667.472  696.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     95  667.472  696.
0.00 100.00 100.00 100.00  94.86  93.15    168   28.822     95  667.472  696.
0.00 100.00 100.00 100.00  94.85  93.14    168   28.822     96  674.816  703.
0.00 100.00 100.00 100.00  94.85  93.14    168   28.822     96  674.816  703.
0.00 100.00 100.00 100.00  94.85  93.14    168   28.822     96  674.816  703.
0.00 100.00 100.00 100.00  94.85  93.14    168   28.822     96  674.816  703.
0.00 100.00 100.00 100.00  94.85  93.14    168   28.822     96  674.816  703.
0.00 100.00 100.00 100.00  94.85  93.14    168   28.822     96  674.816  703.
0.00 100.00 100.00 100.00  94.85  93.14    168   28.822     96  674.816  703.
```

从上面的数据来看，FullGC 在不断出现，但是又回收不了内存，这个问题就严重了。

你要注意，对于这种情况，我们正常的判断逻辑应该是：**一个实时的业务系统就算是有 FullGC，也应该是每次都回收到正常的状态。如果 HEAP 内存确实不够用，那我们可以增加。但是如果 HEAP 一直在减少，直到 FullGC 也回收不了，那就有问题了。**

因此，对于这样的问题，我们要做两方面的分析：

1. 内存确实在被使用，所以，FullGC 回收不了。
2. 内存有泄露，并且已经泄露完，所以，FullGC 无法回收。

那么接下来，我们在做定向监控分析时就要从这两个角度来思考。

定向监控分析

既然内存已经满了，我们就执行一下 jmap -histo:live 1|head -n 50，来看看占比相对较多的内存是什么：

复制代码

```
[root@svc-mall-order-7fbdd7b85f-ks828 /]# jmap -histo:live 1|head -n 50


 num     #instances         #bytes  class name
----------------------------------------------
   1:      74925020     2066475600  [B
   2:       2675397      513676056  [[B
   3:       2675385       85612320  com.mysql.cj.protocol.a.result.ByteArrayRo
   4:       2675386       42806176  com.mysql.cj.protocol.a.MysqlTextValueDeco
   5:        246997       27488016  [C
   6:         80322       16243408  [Ljava.lang.Object;
   7:         14898        7514784  [Ljava.util.HashMap$Node;
   8:        246103        5906472  java.lang.String
   9:        109732        3511424  java.util.concurrent.ConcurrentHashMap$Nod
  10:         37979        3342152  java.lang.reflect.Method
  11:         24282        2668712  java.lang.Class
  12:         55296        2654208  java.util.HashMap
  13:         15623        2489384  [I
  14:         81370        1952880  java.util.ArrayList
  15:         50199        1204776  org.apache.skywalking.apm.agent.core.conte
  16:         36548        1169536  java.util.HashMap$Node
  17:           566        1161296  [Ljava.util.concurrent.ConcurrentHashMap$N
  18:         28143        1125720  java.util.LinkedHashMap$Entry
  19:         13664        1093120  org.apache.skywalking.apm.agent.core.conte
  20:         23071         922840  com.sun.org.apache.xerces.internal.dom.Def
  21:         35578         853872  java.util.LinkedList$Node
  22:         15038         842128  java.util.LinkedHashMap
  23:         52368         837888  java.lang.Object
  24:         17779         711160  com.sun.org.apache.xerces.internal.dom.Def
  25:         11260         630560  com.sun.org.apache.xerces.internal.dom.Def
  26:         18743         599776  java.util.LinkedList
  27:         26100         598888  [Ljava.lang.Class;
  28:         22713         545112  org.springframework.core.MethodClassKey
  29:           712         532384  [J
  30:          6840         492480  org.apache.skywalking.apm.agent.core.conte
  31:          6043         483440  org.apache.skywalking.apm.dependencies.net
  32:          7347         352656  org.aspectj.weaver.reflect.ShadowMatchImpl
  33:          6195         297360  org.springframework.core.ResolvableType
  34:          6249         271152  [Ljava.lang.String;
  35:         11260         270240  com.sun.org.apache.xerces.internal.dom.Att
  36:          3234         258720  java.lang.reflect.Constructor
  37:           390         255840  org.apache.skywalking.apm.dependencies.io.
  38:          7347         235104  org.aspectj.weaver.patterns.ExposedState

```

```
45    39:          5707         228280  java.lang.ref.SoftReference
46    40:          3009         216648  org.apache.skywalking.apm.agent.core.conte
47    41:         13302         212832  org.apache.ibatis.scripting.xmltags.Static
48    42:          8477         203448  org.apache.skywalking.apm.dependencies.net
49    43:          5068         162176  java.util.concurrent.locks.ReentrantLock$N
50    44:          2995         143760  org.apache.skywalking.apm.agent.core.conte
51    45:          2426         135856  java.lang.invoke.MemberName
52    46:          3262         130480  java.util.WeakHashMap$Entry
53    47:          1630         130400  org.apache.skywalking.apm.agent.core.conte
```

在分析内存时，我们可以过滤掉 java 自己的对象，只看和业务相关的对象。从上面的第 3、4 条可以看出，com.mysql.cj.protocol 和 SQL 相关，那我们就到 innodb_trx 表中去查一下，看看有没有执行时间比较长的 SQL。

在查询过程中，我们看到了这样一条 SQL：

复制代码

```
1 select id, member_id, coupon_id, order_sn, create_time, member_username, total
```

进而我又查询了这个语句，发现涉及到的数据有 4358761 条，这显然是代码写的有问题。那我们就去查看一下在代码中，哪里调用了这个 SQL。

通过查看代码，看到如下逻辑：

复制代码

```
1 example.createCriteria().andIdEqualTo(orderId).andStatusEqualTo(0).andDeleteSt
2 List<OmsOrder> cancelOrderList = orderMapper.selectByExample(example);
```

这段代码对应的 select 语句是：

复制代码

```
1  <select id="selectByExample" parameterType="com.dunshan.mall.model.OmsOrderEx
2     select
3     <if test="distinct">
4       distinct
5     </if>
6     <include refid="Base_Column_List" />
7     from oms_order
```

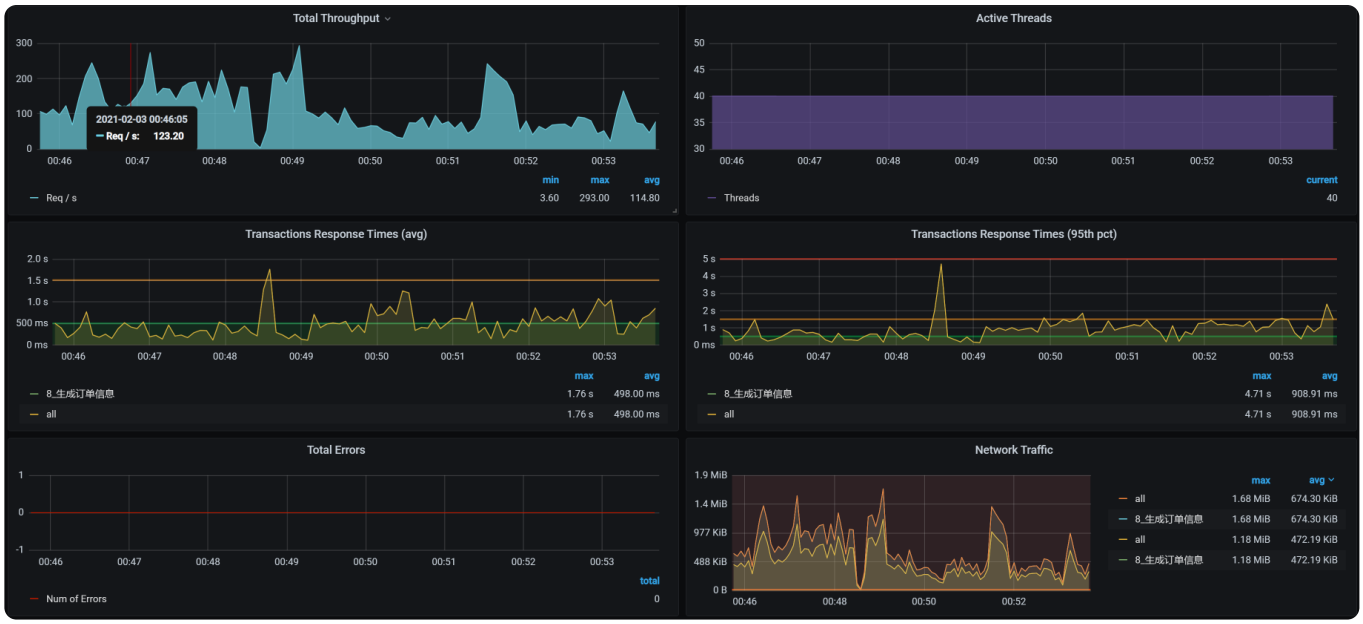
```
    <if test="_parameter != null">
      <include refid="Example_Where_Clause" />
    </if>
    <if test="orderByClause != null">
      order by ${orderByClause}
    </if>

```

这是一个典型的语句没过滤的问题。像这样的开发项目，也最多就是做个 Demo 用。要是在真实的线上项目中，早就不知道伤害了多少人。

我们在这里直接修改代码加上 limit，不让它一次性查询出所有的数据。

然后，我们看一下优化效果：



你看，没有出现 TPS 断裂的情况了，优化效果还是有的，说明那条 SQL 语句不会再查出太多数据把内存给占满了。

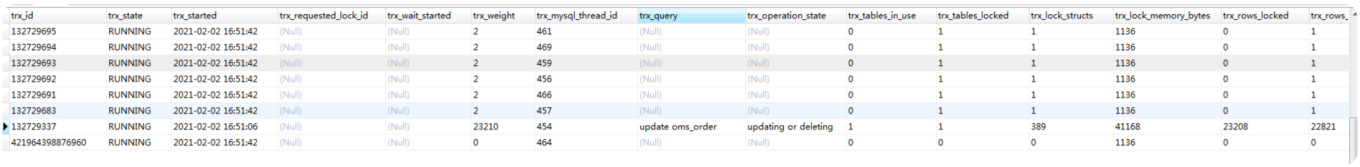
不过，TPS 值并没有增加多少，所以我们必须做第三阶段的分析。

第三阶段

这次我们不从全局监控数据来看了，有了前面的经验，我们直接来做定向监控分析。

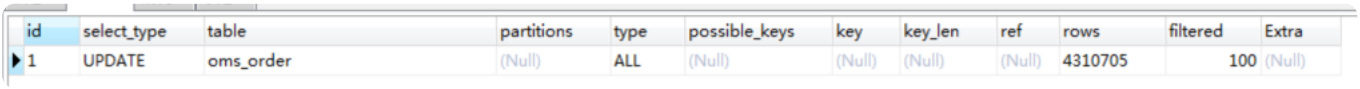
定向监控分析

因为我们在前面改了 SQL，所以在执行 SQL 之后，我们要去查一下 innodb_trx 表，看看还有没有慢的 SQL。 结果，看到了如下 SQL：



trx_id	trx_state	trx_started	trx_requested_lock_id	trx_wait_started	trx_weight	trx_mysql_thread_id	trx_query	trx_operation_state	trx_tables_in_use	trx_tables_locked	trx_lock_structs	trx_lock_memory_bytes	trx_rows_locked	trx_rows_
132729695	RUNNING	2021-02-02 16:51:42	(Null)	(Null)	2	461	(Null)	(Null)	0	1	1	1136	0	1
132729694	RUNNING	2021-02-02 16:51:42	(Null)	(Null)	2	469	(Null)	(Null)	0	1	1	1136	0	1
132729693	RUNNING	2021-02-02 16:51:42	(Null)	(Null)	2	459	(Null)	(Null)	0	1	1	1136	0	1
132729692	RUNNING	2021-02-02 16:51:42	(Null)	(Null)	2	456	(Null)	(Null)	0	1	1	1136	0	1
132729691	RUNNING	2021-02-02 16:51:42	(Null)	(Null)	2	466	(Null)	(Null)	0	1	1	1136	0	1
132729683	RUNNING	2021-02-02 16:51:42	(Null)	(Null)	2	457	(Null)	(Null)	0	1	1	1136	0	1
132729337	RUNNING	2021-02-02 16:51:06	(Null)	(Null)	23210	454	update oms_order	updating or deleting	1	1	389	41168	23208	22821
421964398876960	RUNNING	2021-02-02 16:51:42	(Null)	(Null)	0	464	(Null)	(Null)	0	0	0	1136	0	0

把这个 SQL 拿出来，看看它的执行计划：



id	select_type	table	partitions	type	possible_keys	key	key_len	ref	rows	filtered	Extra
1	UPDATE	oms_order	(Null)	ALL	(Null)	(Null)	(Null)	(Null)	4310705	100	(Null)

又是一个典型的全表扫描，并且是由一个 update 使用的。看到这里，你是不是有种想把开发拉出去祭旗的冲动？

由于生成订单信息是一个复杂的接口，我们不急着收拾这个 SQL，先把 slow log 全都拿出来分析一遍。

请你注意，有时候项目执行的场景多了，数据相互之间的影响就会很大，容易导致我们分析的方向不准确。所以，我们最好把 slow log 都清一遍。反正我通常都会这么干，因为我不想看到乱七八糟的数据。

在清理完慢 SQL、重新执行场景之后，我又把 slow log 拿出来，用 pt-digest-query 分析了一遍（关于这一点，我们在第 16 讲中讲过，如果你不记得的话，建议你再回顾一下），看到如下的数据：

复制代码

```
# Profile
# Rank Query ID                     Response time   Calls R/Call   V/M   I
# ==== ============================ =============== ===== ======== ===== =
#    1 0x2D9130DB1449730048AA1B5... 1233.4054 70.5%     3 411.1351  2.73 UPDAT
#    2 0x68BC6C5F4E7FFFC7D17693A...  166.3178  9.5%  2677   0.0621  0.60 INSER
#    3 0xB86E9CC7B0BA539BD447915...   91.3860  5.2%  1579   0.0579  0.01 SELEC
#    4 0x3135E50F729D62260977E0D...   61.9424  3.5%     4  15.4856  0.30 SELEC
#    5 0xAE72367CD45AD907195B3A2...   59.6041  3.4%     3  19.8680  0.13 SELEC
#    6 0x695C8FFDF15096AAE9DBFE2...   49.1613  2.8%  1237   0.0397  0.01 SELEC
#    7 0xD732B16862C1BC710680BB9...   25.5382  1.5%   471   0.0542  0.01 SELEC
# MISC 0xMISC                       63.2937  3.6%  1795   0.0353   0.0 <9 IT
```

通过上面的 Profile 信息我们看到，第一个语句消耗了总时间的 70.5%，第二个语句消耗了总时间的 9.5%。我们说要解决性能问题，其实解决的就是这种消耗时间长的语句。而后面的 SQL 执行时间短，我们可以暂时不管。

通常在这种情况下，你可以只解决第一个语句，然后再回归测试看看效果，再来决定是否解决第二个问题。我先把这两个完整的 SQL 语句列在这里：

```
1. UPDATE oms_order SET member_id = 260869, order_sn = '202102030100205526', c


2. insert into oms_order (member_id, coupon_id, order_sn,   create_time, membe
```

我们先来看第一个语句。这个 update 语句虽然被调用的次数不多，但是特别慢。这显然不应该是实时接口在调用，那我们就要查一下到底是什么业务调用了这个语句。你看，在这个语句中，update 更新的是 where 条件中 ID 为 0 的数据，这看上去就是一个批量业务。

我们再来看第二个语句。第二个 insert 语句调用次数多，应该是实时交易的 SQL。通常，我们会通过批量插入数据来优化 insert，所以，就需要调整 bulk_insert_buffer_size 参数（默认是 8M）来实现这一点。因为 bulk_insert_buffer_size 就是在批量插入数据时提高效率的。我去查询了一下这个参数，确实没有优化过，还是默认值。

这里你要注意一点，在生产环境中，因为 Order 表中要加索引，所以在架构设计时也最好是主从分离，让 update、insert 和 select 不会相互影响。

分析完这两个 SQL 语句，我们先来查找第一个 SQL 的来源。通过查找代码，可以看到这里调用了该语句：

```
orderMapper.updateByPrimaryKeySelective(cancelOrder);
```

但是，请注意，这个 updateByPrimaryKeySelective 方法是批量任务中的，而批量任务应该和实时交易分开才是。如果你是作为性能团队的人给架构或开发提优化建议，那你可以

这样给建议：

1. 读写分离；

2. 批量业务和实时业务分离。

在这里，我先把这个批量业务给分离开，并且也不去调用它。但是，在真实的业务逻辑中，你可不能这么干。我之所以这么做，是为了看后续性能优化的效果和方向。

做了上述修改之后，TPS 如下：

从效果上来看，TPS 能达到 300 左右了，响应时间看起来也稳定了。我们终于可以进入正常的性能分析阶段了，哈哈。

不过，到这里，我们的工作并没有结束，从上图来看，系统资源并没有完全用起来，这个接口显然还有优化的空间。所以，在下节课中，我们接着来唠。

总结

在这节课中，我们做了三个阶段的分析优化。

在第一阶段中，我们修改了 JDBC 池，虽然 TPS 有上升的趋势，但是，新问题也同样出现了：TPS 非常不稳定，还有断断续续的情况。

在第二阶段中，我们分析了内存溢出的问题，定位出了原因并优化了内存问题。虽然我们在 TPS 曲线上明显看到了优化的效果，但仍然没有达到理想的程度。

在第三阶段中，我们分析定位了 SQL 的问题，这是非常合乎逻辑的。因为我们在第二阶段中修改了 SQL，所以到了第三阶段，就要直接到数据库中做相应的定位。从结果上看，效果不错，TPS 已经有明显正常的趋势了。不过，你要注意的是，当批量业务和实时业务同时出现在同一个数据库中，并且是对同样的表进行操作，这时，你就得考虑一下架构设计是否合理了。

总之，在这节课中你可以看到，当 SQL 查询出来的数据到了应用内存的时候，导致了内存的增加。而应用内存的增加也增加了 GC 的次数，进而消耗了更多的 CPU 资源。

课后作业

最后，请你思考两个问题：

1. 为什么 JDK 中看到 VM Thread 线程消耗 CPU 高，会去查看内存消耗是否合理?
2. 在 MySQL 中分析 SQL 问题时为什么要先查询 innodb_trx 表?

记得在留言区和我讨论、交流你的想法，每一次思考都会让你更进一步。

如果你读完这篇文章有所收获，也欢迎你分享给你的朋友，共同学习进步。我们下一讲再见！

提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 18 | 购物车信息确定订单：为什么动态参数化逻辑非常重要?

下一篇 20 | 生成订单信息之二：业务逻辑复杂，怎么做性能优化?

精选留言 (1)

写留言

张东炫
2021-05-06

1.VM Thread 线程消耗 CPU 高的异常，查看JAVA gc 是否合理
2.innodb_trx表提供了当前innodb引擎内每个事务的信息（只读事务除外），包括当一个事务启动，事务是否在等待一个锁，以及交易正在执行的语句

展开